Get Your Optimizer to Give up All Its Secrets

Brian Hansen
brian@tf3604.com
@tf3604
Thank You Sponsors

Platinum  Dell EMC

Gold  Amazon Web Services  Microsoft  CozyROC  Rubrik

Silver  IDERA  Minionware

Bronze  Redgate  PASS  Revgen

Swag  SentryOne

Venue  Colorado Technical University
Brian Hansen

- 15+ Years working with SQL Server
  - Development work since 7.0
  - Administration going back to 6.5
  - Fascinated with SQL internals

brian@tf3604.com
@tf3604.com

children.org
www.tf3604.com/optimizer
About This Session

• What this session is not
  • An end-to-end optimizer session
  • A performance tuning session

• Goals of this session
  • Additional understanding of SQL Server internals
  • Deeper understanding: write better queries!
  • Provide additional skills for performance tuning
Agenda

• Background:
  • Logical processing order
  • Physical processing considerations
• Executing a query: parse, bind, transform, optimize, execute
• Heuristics, transformation rules, parse trees, memos
• Limitations & DMVs
This Is Only the Foundation

• The materials we are covering here will only skim the surface of what is possible.
• Understanding optimizer internals takes time and study.
• Many features you run across have minimal available information out there.
• Don’t get frustrated ... just keep on diving in (if this is interesting to you)!
Logical Processing Order

• Defines the sequence in which SQL elements are logically processed
• Forms the starting basis for parsing the submitted query
• Usually discussed from the perspective of a SELECT query; similar for UPDATE / DELETE / INSERT / MERGE
• Declarative vs procedural programming
  • “What” vs “How”
Logical Processing Order

- FROM
- ON
- JOIN / APPLY
- PIVOT / UNPIVOT
- WHERE
- GROUP BY
- WITH CUBE / ROLLUP
- HAVING
- SELECT
- DISTINCT
- ORDER BY
- TOP
- OFFSET ... FETCH

For more details, see this and subsequent articles from Itzik Ben-Gan
```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 as ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerId
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

<table>
<thead>
<tr>
<th>Table</th>
<th>Columns</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderHeader</td>
<td>3</td>
<td>301,811</td>
</tr>
<tr>
<td>OrderDetail</td>
<td>5</td>
<td>603,133</td>
</tr>
<tr>
<td>Customer</td>
<td>6</td>
<td>70,132</td>
</tr>
</tbody>
</table>
```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 1: FROM
• OrderHeader joined to OrderDetail
• Perform Cartesian join
• Result is 182,032,173,863 rows / 8 columns
• This is result table R1
```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 2: FROM

- R1 joined to Customer (Cartesian join)
- Result is 12,766,280,417,359,916 rows / 14 columns
- This is result table R2
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;

Step 2: ON
• Find rows in R2 where OrderId = OrderId
• Result is 42,298,923,556 rows / 14 columns
• This is result table R3
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders 
from dbo.OrderHeader oh 
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId 
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID 
where cust.State = 'CO' 
group by od.ProductId 
having sum(od.Quantity) >= 20 
order by od.ProductId;

Step 4: ON
- Find rows in R3 where CustomerId = CustomerId
- Result is 603,133 rows / 14 columns
- This is result table R4


```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CI'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 5: WHERE
- Find rows in R4 where State = 'CO'
- Result is 5,044 rows / 14 columns
- This is result table R5
```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 6: GROUP BY

- Arrange rows into groups by ProductId
- Within each group compute SUM(Quantity)
- Result is 5,044 rows / 2 columns
- This is result table R6 (ProductId, SUM(Quantity))
  - Only these 2 columns are available in downstream steps
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;

Step 7: HAVING
• Find rows in R6 where SUM(Quantity) >= 20
• Result is 58 rows / 2 columns
• This is result table R7
```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'

group by od.ProductId

having sum(od.Quantity) >= 20

order by od.ProductId;
```

Step 8: SELECT
- Evaluate expressions in the select list
  - ProductId → ProductId
  - SUM(Quantity) – 20 → ExcessOrders
- Result is 58 rows / 2 columns
- This is result table R8
```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 9: ORDER BY
- Sort R8 by ProductId
- Result is 5 rows / 2 columns
- This is result table R9
```sql
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 10: TOP
- Keep the first 5 rows in R9
- Remaining rows get discarded
- Result is 5 rows / 2 columns
- This is result table R10
- Logical processing is complete
How SQL Server Sees the Query

```
from dbo.OrderHeader oh
inner join dbo.OrderDetail od
inner join dbo.Customer cust
on oh.OrderId = od.OrderId
on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
select od.ProductId, sum(od.Quantity) - 20 ExcessOrders
order by od.ProductId
top 5;
```
Logical Operators

- Get
- Join
  - \( \bowtie \) inner
  - \( \bowtie \bowtie \bowtie \) outer
  - \( \times \) Cartesian
  - \( \bowtie \bowtie \) semi
  - \( \triangleright \) anti-semi
- Apply

- Set Operators
  - \( \cup \) union
  - \( \cap \) intersection
  - \( \setminus \) except
  - \( \sigma \) Select (SQL: where)
  - \( \pi \) Project (SQL: select)
  - \( G \) Aggregate
## Join Type Comparison

<table>
<thead>
<tr>
<th>Join Type</th>
<th>Condition</th>
<th>Left rows</th>
<th>Right rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner</td>
<td>For each predicate match, output left row + right row</td>
<td>0+</td>
<td>0+</td>
</tr>
<tr>
<td>Left outer</td>
<td>Same as inner, but if no predicate match, output left row + NULL placeholders for right row</td>
<td>1+</td>
<td>0+</td>
</tr>
<tr>
<td>Full outer</td>
<td>Same as left, but if no predicate match in right, output NULL placeholders for left table + right row</td>
<td>1+</td>
<td>1+</td>
</tr>
<tr>
<td>Cartesian / cross (m \times n)</td>
<td>Match each row in left with each row in right (no concept of predicate)</td>
<td>n</td>
<td>m</td>
</tr>
<tr>
<td>Left semi</td>
<td>Output left row once if predicate match</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>Left anti-semi</td>
<td>Output left row once if no predicate match</td>
<td>0 or 1</td>
<td>0</td>
</tr>
</tbody>
</table>
Physical Operators: Logical “get”

- Scan
- Seek
- Lookups
- Heap vs clustered index vs non-clustered index
- Ordered vs unordered
- Forward vs backward
Physical Operators: Other

- Join
  - Merge
  - Nested loops
  - Hash

- Aggregate
  - Stream aggregate
  - Hash aggregate

- Select
  - Filter
Process of Executing a Query

- Parsing and binding
- Optimization
- Execution
Process of Executing a Query - Graphical

```
select top 5 od.ProductId, sum(od.Quantity) - 20 as ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od
on oh.OrderId = od.OrderId
inner join dbo.Customer cust
on oh.CustomerId = cust.CustomerId
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```
• Algebrizer (the “normalizer” in SQL 2000)
  • Parser: validate syntactical correctness
    • Build initial parse tree
    • Identify constants
    • Check user permissions

```
select col1
from objA
where col2 = 1
order by col3
```
select c.CustomerID, c.FirstName, c.LastName, oh.OrderDate 
from CorpDB.dbo.ImportantCustomers c 
inner join CorpDB.dbo.OrderHeader oh on oh.CustomerId = c.CustomerId 
where c.LastName = 'Hansen';

select c.CustomerID, c.FirstName, c.LastName, oh.OrderDate 
from 
  ( 
    select c.CustomerId, c.FirstName, c.LastName, c.State 
    from CorpDB.dbo.Customer c 
    where c.State = 'CO' 
  ) c 
inner join CorpDB.dbo.OrderHeader oh on oh.CustomerId = c.CustomerId 
where c.LastName = 'Hansen';
Parsing and Binding  (3 of 3)

• Binding
  • Metadata discovery / name resolution
  • Data type resolution (i.e., UNION)
    
    select 1 union all select 'Some text';
    Conversion failed when converting the varchar value 'Some text' to data type int.

• Aggregate binding
  
  select LastName, CustomerID, count(*) Nbr from Customer group by LastName;
  Column 'Customer.CustomerID' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.
Parse Trees*

• Internal representation of query operation
• Nodes may be logical or physical operators
  • 0 to infinity inputs, 1 output
• SQL Server will output parse trees at various phases of optimization
  • A variety of trace flags will trigger output

* Or query trees, or relational trees
select top 5
    od.ProductId,
    sum(od.Quantity) - 20 ExcessOrders
from CorpDB.dbo.OrderHeader oh
inner join CorpDB.dbo.OrderDetail od
    on oh.OrderId = od.OrderId
inner join CorpDB.dbo.Customer cust
    on oh.CustomerId = cust.CustomerID
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
from dbo.OrderHeader oh
inner join dbo.OrderDetail od
inner join dbo.Customer cust
on oh.OrderId = od.OrderId
on oh.CustomerId = cust.CustomerId
where cust.State = 'CO'
group by od.ProductId
having sum(od.Quantity) >= 20
select od.ProductId,
    sum(od.Quantity) - 20 ExcessOrders
order by od.ProductId
top 5;
Logical Plans

- Similar to physical execution plans
- Multiple logical plans generated during query optimization
- Have no physical properties, such as
  - Indexes
  - Row counts
  - Keys
- Logical operators only
## Showing Query Trees

<table>
<thead>
<tr>
<th>Trace Flag</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3604</td>
<td>Output extra information to “Messages” tab in SSMS</td>
</tr>
<tr>
<td>8605</td>
<td>Show initial parse tree (converted)</td>
</tr>
<tr>
<td>8606</td>
<td>Show transformed parse trees (input, simplified, join-collapsed, normalized)</td>
</tr>
<tr>
<td>8607</td>
<td>Show output tree</td>
</tr>
</tbody>
</table>
Optimization (1 of 2)

- Simplification (heuristic rewrites, not cost-based)
  - Standardize queries, remove redundancies
    - Subqueries to joins
    - Predicate pushdown
    - Foreign key table removal
  - Contradiction detection
  - Aggregates on unique keys
  - Convert outer join to inner

- Retrieve statistics; do cardinality estimation
  - Create / update auto stats
  - SQL Server 7 vs 2014/2016 CE engine
  - Other physical properties (keys, nullability, constraints)

- Trivial plan
  - Only one possible way to execute query
Optimization (2 of 2)

- Search phases 0 through 2
  - Search 0: “Transaction Processing”
    - Simple, basic tests; internal cost threshold
  - Search 1: “Quick Plan”
    - More rules, parallel exploration; internal cost threshold
  - Search 2: “Full Optimization”
    - Full set of rules; usually exits on timeout
    - Extensive use of heuristics to prune search space
- Construct execution plan
- Plan caching (query text hash, set options)
Search Space

• “Every possible execution plan that achieves the directive of a given query”
• Can be an enormous number of plans
• Consider:
  
  ```
  select ...
  from a join b on ... join c on ... join d on ...
  ```

• Assume a, b, c, d are tables with clustered index & 3 non-clustered indexes each
```
select ...
from a join b on ... join c on ... join d on ...
```

<table>
<thead>
<tr>
<th>Physical access methods (per table)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered clustered index scan</td>
<td>1</td>
</tr>
<tr>
<td>Unordered nonclustered index scan (covering)</td>
<td>3</td>
</tr>
<tr>
<td>Ordered clustered index scan</td>
<td>1</td>
</tr>
<tr>
<td>Ordered nonclustered index scan (covering)</td>
<td>3</td>
</tr>
<tr>
<td>Nonclustered seek + ordered partial scan + lookup</td>
<td>3</td>
</tr>
<tr>
<td>Unordered nonclustered index scan + lookup</td>
<td>3</td>
</tr>
<tr>
<td>Clustered index seek + ordered partial scan</td>
<td>1</td>
</tr>
<tr>
<td>Nonclustered index seek + ordered partial scan (covering)</td>
<td>3</td>
</tr>
<tr>
<td>Indexed views</td>
<td>0</td>
</tr>
<tr>
<td>Index intersection*</td>
<td>54</td>
</tr>
</tbody>
</table>

| Total                              | 72    |

---

6 combinations of 2 indexes; 1 join per pair = 6 joins; 3 join methods each = 18
6 combinations of 3 indexes; 2 joins per triplet = 12 joins; 3 join methods each = 36; total = 54
select ...  
from a join b on ... join c on ... join d on ...

<table>
<thead>
<tr>
<th>Logical Join Orders: 24 Total (or are there more?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a⋈b⋈c⋈d</td>
</tr>
<tr>
<td>a⋈b⋈d⋈c</td>
</tr>
<tr>
<td>a⋈c⋈b⋈d</td>
</tr>
<tr>
<td>a⋈c⋈d⋈b</td>
</tr>
<tr>
<td>a⋈d⋈c⋈b</td>
</tr>
<tr>
<td>a⋈d⋈b⋈c</td>
</tr>
<tr>
<td>b⋈a⋈c⋈d</td>
</tr>
<tr>
<td>b⋈a⋈d⋈c</td>
</tr>
<tr>
<td>b⋈c⋈a⋈d</td>
</tr>
<tr>
<td>b⋈c⋈d⋈a</td>
</tr>
<tr>
<td>b⋈d⋈a⋈c</td>
</tr>
<tr>
<td>b⋈d⋈c⋈a</td>
</tr>
</tbody>
</table>
Join Order Considerations

- So far we’ve only considered “left-deep” trees
- \( n! \)
Join Order Considerations, continued

- There are also “bushy” trees
- \((2n-2)!/(n-1)!\)
- Optimizer normally does not consider these
select ...
from a join b on ...
join c on ...
join d on ...

- 72 possible physical data access methods
- 120 possible logical join orders
- 3 physical joins possible per logical join
  - May require intermediate sort operation
- = 25,920 possible plans
- Much larger for more complex queries
- Optimizer uses heuristics to limit search space
sys.dm_exec_query_optimizer_info

• Documented. Sort of.
• Three columns:
  • counter: Name of the observation
  • occurrence: Number of times observation was recorded
  • value: Average per occurrence
• Collect before and after images of this view on a quiet system
Smart Optimization

http://imgs.xkcd.com/comics/efficiency.png
Can the Optimizer Dig a Bit Deeper?

- Trace flag 8780:
  - Considerably more attempts in Search 2
  - Very often still won’t come up with a different (or better) plan
An Interesting Metric: Gain

- Indicates improvement from phase to phase
  - Search 0 to 1 gain
  - Search 1 to 2 gain
  - Value that is $\geq 0$ and $< 1$
    - 0 indicates no improvement
    - Approaching 1 indicates significant improvement

- Definition:

$$\text{Gain}_{S_0 \text{ to } S_1} = \frac{\text{MinCost}(S_0) - \text{MinCost}(S_1)}{\text{MinCost}(S_0)}$$
Heuristics and Transformations

• Heuristics
  • Rules that can eliminate entire branches of the search space

• Transformations
  • Find equivalent operations to get same output
  • Rule-based
    • DBCC SHOWONRULES
    • DBCC RULEON / RULEOFF
  • Four types
    • Simplification, exploration, implementation, property enforcement
Transformations: Exploration

• Start from a logical operation (may be a sub-branch of the full query): the pattern
• Find equivalent logical operations: the substitute
• Examples:
  • Join commutativity: $A \bowtie B \rightarrow B \bowtie A$
  • Join associativity: $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$
  • Aggregate before join
Transformations: Implementation

• Start from a logical operation
• Find equivalent physical operation
• Example:
  • $A \bowtie B \rightarrow A$ (nested loops join) $B$
  • $A \bowtie B \rightarrow A$ (merge join) $B$
  • $A \bowtie B \rightarrow A$ (hash join) $B$
• Obtain costing on physical operations
• Can prune expensive branches from tree
Transformations: Property Enforcement

• Properties associated with parse tree nodes
  • Uniqueness, type, nullability, sort order
  • Constraints on column values

• Transformation rules may cause certain properties to be enforced
  • Example: sort order for a merge join
sys.dm_exec_query_transformation_stats

- One row per transformation rule
  - “Promise_Total” – Estimate of how useful might the rule be for this query
  - “Built_Substitute” – Number of times the rule generated an alternate tree
  - “Succeeded” – Number of times the rule was incorporated into search space
- Collect before and after images of this view on a quiet system
Factors Considered by the Optimizer

• Memory grants
• Costing
  • Cold cache
  • Sequential vs random I/O
    • But not the nature of the I/O subsystem
  • CPU costs, core count, available memory
  • Cardinality estimator
• What do cost units really mean?
Memo Structure

• Used to explore different alternatives to a portion of the query tree
• Can think of it as a matrix
  • Rows (groups) represent substitutes – each entry is logically equivalent
  • Columns represent application of a transformation rule
• Each entry is hashed to prevent duplication
• Physical substitutes are costed
Example Memo

```
select *
from CorpDB.dbo.OrderDetail od
inner join CorpDB.dbo.OrderHeader oh on od.OrderId = oh.OrderId
inner join CorpDB.dbo.Customer c on c.CustomerID = oh.CustomerId;
```

<table>
<thead>
<tr>
<th>Group</th>
<th>Option 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Join 2 ⋈ 3</td>
</tr>
<tr>
<td>3</td>
<td>Get Customer</td>
</tr>
<tr>
<td>2</td>
<td>Join 0 ⋈ 1</td>
</tr>
<tr>
<td>1</td>
<td>Get OrderHeader</td>
</tr>
<tr>
<td>0</td>
<td>Get OrderDetail</td>
</tr>
</tbody>
</table>
Example Memo

- Apply join associativity:
  - \((OD \bowtie OH) \bowtie C \rightarrow OD \bowtie (OH \bowtie C)\)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option x.0</th>
<th>Option x.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.x</td>
<td></td>
<td>Join 1.0 \bowtie 3.0</td>
</tr>
<tr>
<td>4.x</td>
<td>Join 2.0 \bowtie 3.0</td>
<td>Join 0.0 \bowtie 5.1</td>
</tr>
<tr>
<td>3.x</td>
<td>Get Customer</td>
<td></td>
</tr>
<tr>
<td>2.x</td>
<td>Join 0.0 \bowtie 1.0</td>
<td></td>
</tr>
<tr>
<td>1.x</td>
<td>Get OrderHeader</td>
<td></td>
</tr>
<tr>
<td>0.x</td>
<td>Get OrderDetail</td>
<td></td>
</tr>
</tbody>
</table>
Example Memo

- Apply join commutativity:
  - 

<table>
<thead>
<tr>
<th>Group</th>
<th>Option x.0</th>
<th>Option x.1</th>
<th>Option x.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.x</td>
<td></td>
<td>Join 1.0 ⊙ 3.0</td>
<td></td>
</tr>
<tr>
<td>4.x</td>
<td>Join 2.0 ⊙ 3.0</td>
<td>Join 0.0 ⊙ 5.1</td>
<td>Join 2.2 ⊙ 3.0</td>
</tr>
<tr>
<td>3.x</td>
<td>Get Customer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.x</td>
<td>Join 0.0 ⊙ 1.0</td>
<td></td>
<td>Join 1.0 ⊙ 0.0</td>
</tr>
<tr>
<td>1.x</td>
<td>Get OrderHeader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.x</td>
<td>Get OrderDetail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Optimizer Is Exceptionally Complex

- It has to deal with many things we’ve not discussed
  - DML (updates, deletes, inserts, merges; output clause)
    - Halloween protection
    - Triggers
    - Index updates
    - Constraint management
  - Wide vs. narrow updates
  - Data warehouse optimization
  - Columnstore, full-text, spatial, xml, filtered indexes and sparse columns
  - Window functions, partitioned tables, Hekaton, Stretch DB, other new features
  - Row vs. batch mode
  - And much more
Conclusions

• SQL is a declarative language
  • In theory, it shouldn’t matter how SQL is written
    • We are effectively giving SQL Server a set of requirements and asking it to write a program for us
  • In practice, it does matter because no optimizer is perfect
    • It will give us correct results
    • In the real world, efficiency matters

• Writing “better” queries
  • Sometimes we need to “out-smart” the optimizer
## Appendix: Trace Flags

<table>
<thead>
<tr>
<th>TF</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2363</td>
<td>Show statistics used by optimizer (SQL 2014+ CE) and <em>lots</em> of other info</td>
</tr>
<tr>
<td>2372</td>
<td>Show memory usage at each phase</td>
</tr>
<tr>
<td>2373</td>
<td>Show memory usage for rules and properties</td>
</tr>
<tr>
<td>3604</td>
<td>Output to client (&quot;Messages&quot; tab)</td>
</tr>
<tr>
<td>7352</td>
<td>Show final query tree (post-optimization rewrites)</td>
</tr>
<tr>
<td>8605</td>
<td>Show initial parse tree (converted)</td>
</tr>
<tr>
<td>8606</td>
<td>Show transformed parse trees (input, simplified, join-collapsed, normalized)</td>
</tr>
<tr>
<td>8607</td>
<td>Show output tree</td>
</tr>
<tr>
<td>8608</td>
<td>Show initial memo structure</td>
</tr>
</tbody>
</table>
## Appendix: Trace Flags, continued

<table>
<thead>
<tr>
<th>TF</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8609</td>
<td>Show task and operation type counts</td>
</tr>
<tr>
<td>8612</td>
<td>Add cardinality info to tree</td>
</tr>
<tr>
<td>8615</td>
<td>Show final memo structure</td>
</tr>
<tr>
<td>8619</td>
<td>Show applied rules (SQL 2012+)</td>
</tr>
<tr>
<td>8620</td>
<td>Show applied rules and memo arguments (SQL 2012+)</td>
</tr>
<tr>
<td>8621</td>
<td>Show applied rules and resulting tree (SQL 2012+)</td>
</tr>
<tr>
<td>8649</td>
<td>Force parallel plan</td>
</tr>
<tr>
<td>8666</td>
<td>Add debugging info to query plan (in the “F4” properties)</td>
</tr>
<tr>
<td>8675</td>
<td>Show optimization search phases and times</td>
</tr>
<tr>
<td>8757</td>
<td>Disable trivial plan generation</td>
</tr>
</tbody>
</table>
### Appendix: Trace Flags, continued

<table>
<thead>
<tr>
<th>TF</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8780</td>
<td>Give query processor more “time” to optimize query</td>
</tr>
<tr>
<td>9130</td>
<td>Show pushed predicate</td>
</tr>
<tr>
<td>9204</td>
<td>Show statistics used by optimizer (fully loaded) (SQL 7 CE only)</td>
</tr>
<tr>
<td>9292</td>
<td>Show statistics used by optimizer (header only) (SQL 7 CE only)</td>
</tr>
</tbody>
</table>

* And many more ...
Appendix: Commands

- DBCC TRACEON / TRACEOFF
- DBCC RULEON / RULEOFF
- DBCC SHOWONRULES
- DBCC SHOWOFFRULES
- option (recompile, querytraceon ####, queryruleoff ‘xxx’)
- sys.dm_exec_query_optimizer_info
- sys.dm_exec_query_transformation_stats
Appendix: History of SQL’s Optimizer

- **Volcano Optimizer** (April 1993) ([PDF](#))
  - Goetz Graefe, William J. McKenna
  - Based on Graefe’s earlier Exodus Optimizer
- **Cascades Framework** (1995) ([PDF](#))
  - Goetz Graefe
  - Refinement of the Volcano Optimizer
  - Basis for rewritten optimizer in SQL Server 7.0
  - Major innovation: the memo structure
References

• Benjamin Nevarez (Blog)
  • *Inside the SQL Server Query Optimizer*
• Paul White
  • Page Free Space blog (especially this series)
  • SQL Performance blog
• Conor Cunningham (Blog)
  • *Microsoft SQL Server 2012 Internals* (Kalen Delaney, editor), chapter 11
  • SQLBits Session
This presentation and supporting materials can be found at www.tf3604.com/optimizer.

- Slide deck
- Scripts
- Sample database
- SQL Server Query Tree Viewer binaries & source

brian@tf3604.com • @tf3604